Санкт-Петербургский государственный университет
Кафедра компьютерного моделирования и многопроцессорных систем

Базарнов Глеб Сергеевич

Выпускная квалификационная работа бакалавра

Разработка системы поиска по базе документов

Направление 010300
Фундаментальная информатика и информационные технологии

Научный руководитель
PhD
доцент
Корхов В.В.

Санкт-Петербург
2016

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время количество доступной информации постоянно увеличивается. С огромной скоростью растут различного рода информационные сети, и, соответственно, растет и количество данных, доступных пользователю. Постоянно появляются все новые статьи, книги и различные медиафайлы. Конечно же, со временем растет и качество этих данных, что влияет на их итоговый объем. Так, например, с ростом качества изображений, увеличивается и их объем.

В процессе обучения так же используется огромное количество всевозможных материалов. Естественным образом возникает вопрос о их долгосрочном хранении. Также актуальной задачей является вопрос качественного поиска среди доступных файлов. Так, на данный момент, на кафедре файлы хранятся не систематизировано, находятся на различных физических носителях, могут иметь совершенно разные форматы. С течением времени количество этих файлов неуклонно растет, растет и проблема поиска необходимой информации.

Для разрешения этой проблемы необходимо рассмотреть различные технологии предназначенные для хранения информации. Перед нами встанет несколько задач, среди которых будет вопрос и о быстром и качественном поиске по реализованному хранилищу.

ПОСТАНОВКА ЗАДАЧИ

Предпосылки для разработки

На кафедре компьютерного моделирования и многопроцессорных систем скопилось огромное количество различной технической литературы. В документах этой литературы хранится большое количество актуальных и полезных знаний. Проблема заключается в том, что эти данные хранятся бессистемно, причем на различных информационных носителях. Ввиду того, что работа с такими данными становится затруднительной, по мере роста их количества, было принято решение о создании некоторого программного средства для оптимизации проблемы с ними.

Постановка задачи на выпускную квалификационную работу.

В рамках написания выпускной квалификационной работы было необходимо разработать поисковый модуль уже существующей системы хранения файлов.

Поиск по документам осуществляется на основании их содержимого. Необходимо рассмотреть некоторые факторы:

- поиск по фразам;
- нечеткий поиск;
- поиск с использованием метасинтаксической переменной.

Функционал конечного продукта должен удовлетворять следующим требованиям:

1. индексация загруженных документов;

2. поиск документов по выбранным атрибутам;
3. полнотекстовый поиск по содержимому документов;
4. настройка параметров поиска, индексации и кластеризации.

В конечном итоге поисковый модуль конечного программного продукта должен предоставить возможность быстрого и эффективного поиска по системе.

Глава 1 Система хранилища данных

1.1 Архитектура системы

Описываемый программный продукт имеет многоуровневую архитектуру. Подобная архитектура является клиент-серверной и имеет раздельные процессы обработки, управления и представления.

Как наиболее популярная разновидность многоуровневой архитектуры, представлена архитектура, состоящая из трех уровней:

- Клиентский уровень, являющийся пользовательским компонентом;
- Связующий уровень, контролирующий поток данных;
- Уровень данных, к которому относятся базы данных и прочие средства, предназначенные для хранения данных.

Многоуровневая архитектура имеет схожие принципы с трехуровневой. Большее количество слоев может быть результатом деления логики предметной области и приложений, её реализующих, на несколько частей. Полученные части могут выполняться на разных узлах, что приведет к уменьшению степени нагрузки.

Серверная часть комплекса представляет из себя шесть различных модулей:

- Warehouse, отвечающий за хранение данных;
- Auth, осуществляющий авто;
- Data, содержащий информацию о пользователях и их действиях;
- Search, ответственный за выполнение поисковых запросов;
- Web-front. Обеспечивает связь всех остальных модулей с пользователем. Предоставляет браузерный интерфейс в виде HTML-страниц.

- Web-rest. Реализует возможность асинхронных запросов, что позволяет создать динамичный интерфейс.

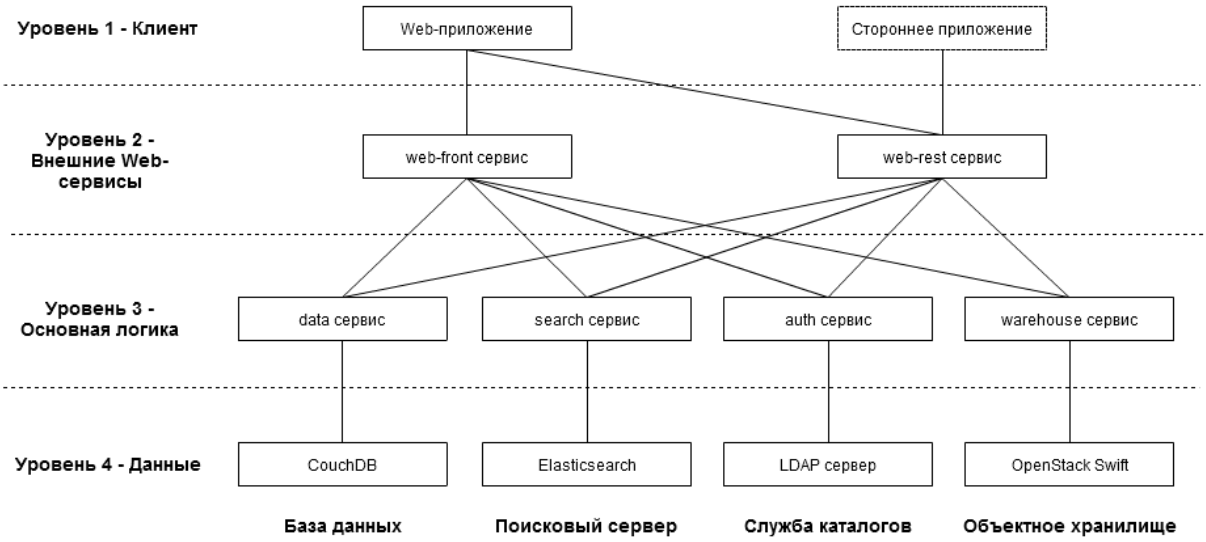


Рисунок 1. Архитектура комплекса

Ввиду разбиения архитектуры на вышеперечисленные модули, появляется возможность реализовать отдельные модули с помощью иных программных комплексов. В тоже время это облегчает масштабирование предоставленного комплекса.

На уровне данных используется нижеперечисленное программное обеспечение

- CoachDB;
- Elasticsearch;
- Openstack Keystone;
- Openstack Swift.

1.2 Структура приложения

Разрабатываемая система имеет модульную структуру с родительскими и дочерними файлами проектов. Для каждого элемента, существующего в данной иерархии есть pom.xml файл, который необходим для конечной сборки при помощи Apache Maven.

В domain-model находятся классы, отвечающие за описание сущностей, которые необходимы для решения определенных задач . Данный модуль включается в модели остальных серверов, отвечающих за авторизацию, хранение и загрузку данных. На базе этих классов реализуются конечные приложения и библиотеки, позволяющие утилизировать разработанные сервисы. Помимо этого, они описывают вспомогательные сущности, необходимые для разрешения некоторых особых проблем, характерных для данной ветки. Данная особенность позволяет использовать интерфейсы для реализации однотипного функционала, используя любое подходящее программное обеспечение. Таким образом, например, в сервере данных можно было использовать MongoDB, вместо CouchDB.

Ветка Core предоставляет реализацию расширенного функционала веб-сервисов. Как видно из диаграммы, эта ветка не зависит от domain-model. Предоставлен аналогичный метод построения модель-приложение-клиен и функционал, отслеживающий состояния системы.

Ветка Web реализует необходимый пользовательский функционал. Модуль web-core отвечает за работу внутренней сети, а остальные модули ветки ответственны за сообщением с этим модулем и предоставлении пользовательских интерфейсов в виде HTML и JSON, для модулей web-front и web-rest соответственно.

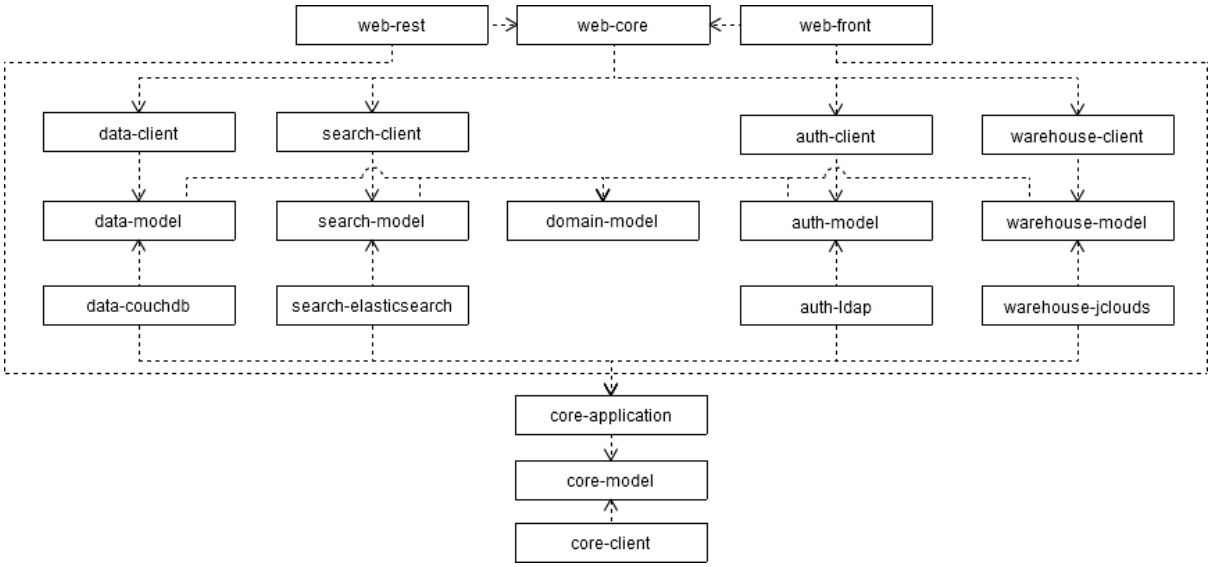


Рисунок 2. Модульная структура системы

1.3 Apache Maven

Apache Maven — это фреймворк, служащий для автоматической сборки проектов. Этот фреймворк описывает структуру файлов на POM языке. В свою очередь, POM является подмножеством расширенного языка разметки XML.[1]

POM файл описывает структуру файла тегами. Тегом, описывающим главную информацию о проекте, является тег <project>, в теле которого содержатся groupId, artifactId, version (имя организации, имя самого проекта и версия соответственно). Далее используется тег, описывающий зависимости проекта — <dependencies>. В нем хранится информация о библиотеках, которые необходимы для успешного построения проекта. Кроме того, для них также указываются имена организации, проекта и их версия.

Для описания возможных используемых плагинов и информации о местоположении файлов можно использовать тег <build>, который является необязательным.

Основной жизненный цикл проекта (default), собранного Maven включает в себя следующие фазы:

- Validate — проверка структуры на полноту и правильность;
- Compile — компиляция исходников;

- Test — тестирование кода набором тестов, заданных заранее;
- Package — процесс упаковки классов в JAR;
- Integration-test — интеграционное тестирование отдельных крупных модулей или всего программного обеспечения;
- Install — занесение проекта в локальный репозиторий. Данный проект становится доступным для остальных проектов пользователя;
- Deploy — копирование проекта в удаленный репозиторий.

Также в Maven содержатся жизненные циклы очистки проекта (clean) и для создания документации (site).

1.4 Apache CouchDB

CouchDB является документо-ориентированной системой управления базами данных. В данной СУБД используется NoSQL подход. Отдельно следует выделить несколько полезных особенностей этой системы,

- MapReduce принцип, который позволяет обрабатывать данные из действительно больших выборок;
- REST интерфейс, т.е. все операции организованы через HTTP;
- Хранение данных в формате JSON;
- Множественная реплекация.

1.5 Компоненты OpenStack

OpenStack — комплекс программных средств для построения облачных сервисов и хранилищ. В комплексе содержится множество различных проектов, способных строить различные компоненты облачных решений. [1]

OpenStack Swift представляет из себя распределенное хранилище. Как сильную сторону следует отметить высокую отказоустойчивость системы. Стоит отметить, что данная система легко поддаётся масштабированию.

Главные компоненты OpenStack Swift:

- Proxy Server — служит для объединения всех компонент. Принимает HTTP-запросы от пользователей и отвечает за поиск объектов;
- Object Server — используется для хранения данных;
- Container Server — выдаёт список объектов;
- Account Server — отдача имен контейнеров для определенных аккаунтов.

OpenStack Keystone — компонент комплекса OpenStack, отвечающий за идентификацию. Данный компонент ответственен за распределение возможностей пользователей и предоставляет им различные роли, такие, как: Searcher, User, Admin.

1.6 Elasticsearch

Elasticsearch — поисковой сервер, являющийся мощным и гибким средством для организации полнотекстового поиска. Изначально основанный на Java библиотеке Apache Lucene, является оптимальным средством для реализации поискового модуля существующей системы хранилища документов.

Подробнее про ElasticSearch и сравнение его с другими поисковыми серверами в следующей главе.

Глава 2 Поисковые машины

2.1 Общее описание поисковых машин

Поисковая машина — программное обеспечение для информационного поиска.

Основной оценкой результата работы, производимой поисковой системой, можно назвать релевантность поиска, проверка морфологии языка и полноту индекса. Под понятием релевантности понимается целесообразность результата, то есть уровень соответствия самого запроса и результата.

Условно поисковые машины можно классифицировать по области поиска на две группы:

1. Поисковые машины локального поиска. Это такие машины, которые используются для поиска по определенной части сети. Поиск может осуществлятся как и по нескольким сайтам сети интернет, так и по локальной сети. Системы подобного типа чаще всего используются в корпоративных сетях, а также в различных комплексах электронной коммерции;
2. Поисковые машины глобального типа. Машины данной группы служат для масштабного поиска по всей сети Интернет, либо же по ощутимо большой её части. В пример пользователей машин подобного типа можно отнести такие корпорации, как Google, Bing и т.д. Чаще всего эти системы предоставляют информационный поиск различных форматов. К файлам, пригодным для работы с подобными машинами относятся файлы как и различных графических форматов, так и графические или мультимедийные. Но все-таки текстовый поиск остаётся наиболее востребованным.

Поисковой запрос — определенная символьная последовательность, используемая для поиска необходимой информации. Запрос является базовой информацией для поиска. Вид запроса зависит от организации поисковой машины, а также от формата искомой информации.

Запросы можно разделить на следующие классы:

- Информационные запросы — поиск определенной информации;
- Навигационные запросы — для поиска определенных компаний или, например, сайтов;
- Транзакционные запросы — для совершения каких-либо действий, таких как покупка товара или установка некого программного обеспечения. [3]

Также поисковые машины часто реализуют связные запросы, которые сообщают о связности индексированного веб-графа (например, какие адреса ссылаются на данный URL) [4]

Очевидно, что в разработанной системе хранилища файлов необходимо использовать поисковую машину локального типа, так как подразумевается работа с различной информацией внутри факультета ПМ-ПУ.

2.2 ElasticSearch

Elasticsearch — поисковой сервер, основанный на библиотеке Apache Lucene. Этот сервер обеспечивает мультиарендный[5] полнотекстовый поиск с HTTP web-интерфейсом и отсутствием схемы, что позволяет загружать в него JSON документы. Elasticsearch разработан на языке Java и выпускается под лицензей Apache License.[6]

Apache Lucene, в свою очередь, представляет из себя свободную библиотеку для полнотекстового поиска. Строго говоря, она не представляет из себя полноценный поисковой сервис, а предназначена в основном для

реализации систем поиска. Основные возможности Lucene — индексировать данные и искать по этим построенным индексам. Любое API для составления запросов, кластеризации, ввода данных и т.д. — останется на части программных средств, основанных на этой библиотеке.

Elasticsearch может быть использован для поиска любых типов документов. Он обеспечивает масштабируемый поиск, имеет поиск, близкий к реальному времени и поддержку мультиарендности.

Еще одной особенностью Elasticsearch являеется “шлюз”, который предоставляет долгосрочное сохранение индекса.[7] Таким образом, индекс может быть извлечен из “шлюза” в случае сбоя сервера. Elasticsearch поддерживает GET запросы в режиме реального времени, что также может сделать его использование пригодным в качестве NoSQL хранилища[8], но без поддержки распределенных транзакций.

В данной поисковой машине, при изменении данных в хранилище, логирование ведется сразу на несколько ячеек кластера, что необходимо для повышения отказоустойчивасти. Также это обеспечивает сохранность данных при возможных сбоях в работе сервера.

Анализаторы Elasticsearch

Особо остро при составлении запросов стоит вопрос выбора корректного анализатора. Сам по себе анализатор является подобием конвейера, который состоит из определенных обработчиков:

- Фильтрация символов;
- Токенизатор;
- Фильтр токенов.

Основная задача анализатора — приняв длинный запрос с определенным количеством лишних деталей на входе, получить из него краткую суть и составить список токенов, её отображающих.

Рисунок 3. Схема анализатора

В первую очередь к введенному запросу применяются символьные фильтры. Результат проделанной работы поступает в токенизатор, который является обязательной частью анализатора. В ходе токенизации, из полученного запроса удаляются знаки препинания, запрос разделяется на независимые токены, которые сохраняют исходную форму, сокращаются до основы слова, либо обрабатываются другим методом, в зависимости от выбранного токенизатора. Следующим этапом токены повторно фильтруются, в случае, если это необходимо.

Изначально, в системе присутствуют определенные анализаторы. Однако, если среди них не окажется подходящего, то можно описать необходимые собственные. Образец несдандартного обработчика представлен ниже:

```
index
    analysis                                                    :
      analyzer                                                  :
        myHTMLSnowball                                          :
          type                          :                   custom
          char_filter                   :             [html_strip]
        tokenizer                       :                 standard
          filter : [lowercase,stop,snowball]
```

Рассмотрим работу анализатора на данном примере:

1. Сначала удаляются все обнаруженные теги, т.к. указан фильтр html_strip;
2. Используя стандартный токенизатор, мы разбиваем запрос на токены и удаляем знаки препинания;
3. Полученные данные приводятся к нижнему регистру;
4. Удаление встретившихся стоп-слов;

5. Производится стемминг полученного результата (фильтр snowball).

Нечеткий поиск

Отдельные трудности представляет из себя обработка естественных языков, так как они отличаются большим количеством неточностей. Чаще всего различные системы стараются проводить анализ структуры языков, выделяя некоторые шаблоны, уместные в данном языке. Но система часто встречает запросы, которые нетипичны для существующей морфологии или орфографии языка. В основном подобные ситуации возникают из-за опечаток или просто неграмотности пользователей.

Нечеткий поиск базируется на расстоянии Дамерау-Левенштейна. Это такая мера разницы, которая определяется минимальным количеством операций, необходимых для перевода одной строки в другую. При нечетком поиске изначально используется именно это расстояние.

Стоит отметить, что все данные, помещающиеся в индексы, обязательно обрабатываются анализатором. Таким образом, в индексе хранятся сокращенные версии данных, с минимальным количеством знаков. По этим урезанным данным проходит поиск, и при нечетком поиске могут возникнуть неожиданные результаты.

Если используется фильтр snowball, то слова будут приводиться к своей основе, что довольно проблематично в случае опечатки пользователем. Таким образом, например, слово stemming будет приведено к основе stem, но по этой основе будет пропущено слово stemmingh, так как потребуется более двух операций, для приведения слова к этой основе. Однако этой проблемы можно избежать, если использовать стандартный стеммер и не пользоваться поиском синонимов.

В Elasticsearch есть некоторые способы нечеткого поиска:

- match query & fuzziness option. Совокупность простого запроса и параметра нечткости. Текст анализируется перед началом поиска;

- fuzzy query. Нечеткий поиск по стеммам, перед поиском анализ не проводиться;
- fuzzy like this/fuzzy like this field. Поддерживается анализ веса, что способствует лучшему ранжированию результатов;
- suggesters. Предположения, базирующиеся на нечетких запросах.

Безопасность

В системе изначально присутствует возможность резервного копирования и восстановления. Для создания резервных копий необходимо указать, где они будут храниться:

```
$ curl -XPUT '//localhost:9200/_snapshot/my_Backups' -d '{
    "type": "fs",
    "settings": {
        "location": "/es_backups",
        "compress": true
    }
}'
```

Здесь:

- type — вид хранилища, где хранятся бекапы. Изначально присутствует файловая система. Используя дополнительные плагины, можно организовать хранение в популярных облачных хранилищах;
- location — адрес для сохранения резервных копий;
- compress — параметр, определяющий, проводить ли компрессию данных. На деле компрессируются только метаданные, так что особой пользы данная функция не несет.

Для сохранения копий:

```
$ curl -XPUT
"localhost:9200/_snapshot/examplebackup/examplesnapshot?wait_for_co
mpletion=true"
```

Для восстановления:

```
$ curl -XPOST
"localhost:9200/_snapshot/mybackup/ecamplesnapshot/_restore"
```

Снимки являются инкрементальными. Изначально сохраняется вся резервная копия, при последующих операциях учитывается только различия между разными версиями копий. Если данные содержатся на файловой системе, то для резервного копирования следует указать сетевой диск, доступ к которому будет у всех узлов.

2.3 Сравнительная характеристика различных комплексов

Главным конкурентом Ealsticsearch является Apache Solr, также основанный на библиотеке Lucene. В принципе, сами системы обладают схожим функционалом, но имеют различий:

Характеристика	Elasticsearch	Solr
Поддержка языков	Java, Groovy, PHP, Ruby, Perl, Python, .NET, Javascript	Java
Использование сторонних скриптов	Поддерживается	Не поддерживается
Зависимость	Использует собственные	Использует ZooKeeper

кластера от стороннего ПО	узлы	
Автоматическая ребалансировка узлов	Выполняется по заданным правилам	Не поддерживается
Контроль размещения	параметр _route_	параметр _routing_
Изменения количества партиций	Количество партиций после создания индекса неизменно, однако выполняется копированием в новый индекс	Увеличиваются путем добавления новых партиций, либо разбиением уже существующих
Перемещение партиций в кластере	Свободно перемещает реплики и партиции	Возможно только копированием с последующим удалением оригинала
Поиск узлов	ZenDiscovery или ZooKeeper	Только ZooKeeper
Согласованность	Доступна синхронная/асинхронная модификация реплик	Завершение индексации только после синхронизации всех реплик.
Автоматический выбор узла	Поддерживается	Поддерживается

Таблица 1. Сравнение ElasticSearch и Solr [9]

Как видно из таблицы, приведенные различия не столь существенны. Однако, ElasticSearch определенно лучше подходит для создания системы. К особо выгодным преимуществом можно отнести автоматическую балансировку нагрузки, самостоятельное peer-to-peer объединение в единый кластер, полноценное перемещение партиций на другой узел, а также возможность модификации с помощью скриптовых языков (типа JavaScript etc).

Глава 3 Реализация системы поиска

Для реализации поискового модуля системы хранилища файлов необходимо установить поисковой сервер Elasticsearch версии 2.3.2, наладить его работу с плагином Logstash для корректной работы с хранилищем.

Далее необходимо реализовать классы, получающие пользовательские запросы, производящие поиск по совпадениям и возвращающие полученный результат.

Для ускорения работы поискового модуля необходимо распараллелить процесс выгрузки полученных результатов. В конечном итоге выгрузка результатов и построение страниц должно происходить параллельно, что позволит ускорить процесс получения пользователем итогового результата.

3.1 Элементы MVC

Для реализации модели MVC используется фреймворк Spring, который позволяет применять шаблоны проектирования, необходимые для данного решения. Этот фреймворк предоставляет каркас, который даёт возможность без особых трудностей создать следующие классы компонентов системы:

- Контроллеры для обработки запросов и передачи их в необходимые сервисы;
- Сервисы с реализацией типовых функций, вызываемые контроллерами;
- Классы, формирующие ответ на пользовательские запросы, в дальнейшем объединяющие экземпляры классов и передающие их пользователю.

3.2 Реализация классов

Для реализации поиска необходимо реализовать классы, которые будут обеспечивать обработку полученных запросов и построение по ним результатов. Рассмотрим процесс построения классов на примере полнотекстового поиска по файлам.

Для задач, связанных с полнотекстовым поиском по хранящимся файлам, в прототипе системы были реализованы контроллер WarehousedFileSearcherController и сервис WarehousedFileSearcherService, которые реализуют методы интерфейса RemoteWarehousedFileSearcher посредством классов поискового сервера search-model и классов доменной модели domain-model, описывающих используемые сущности.

В RemoteWarehousedFileSearcher хранится метод, отвечающий за полнотекстовый поиск.

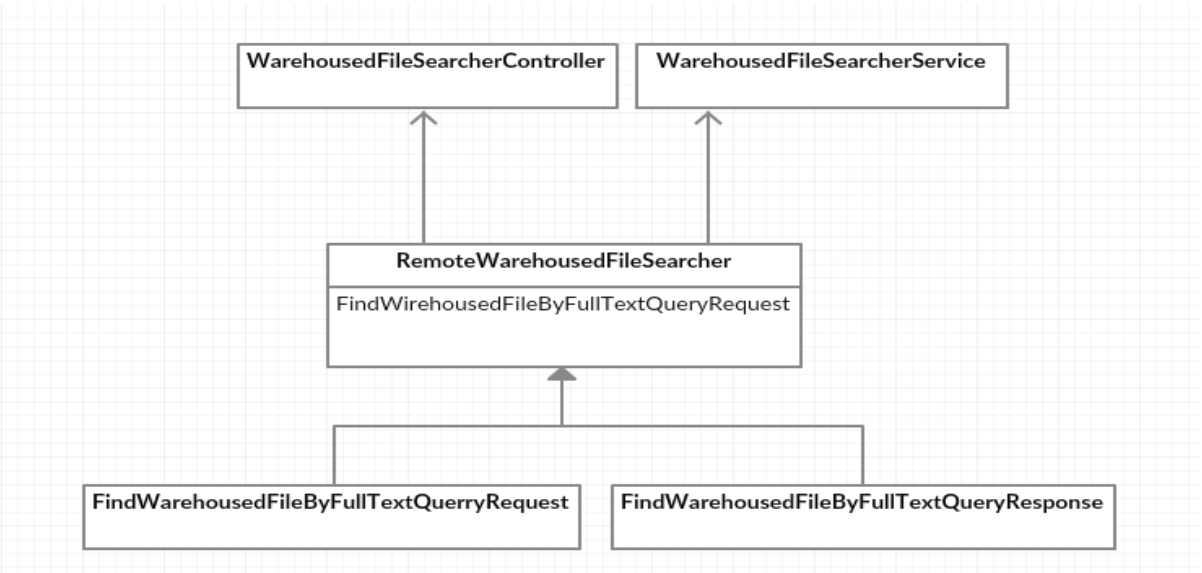


Рисунок 5. Взаимодействие классов полнотекстового поиска

В свою очередь, сервис WarehousedFileSearcherService строит результат при помощи следующих классов:

- Классы из domain-model, описывающие сущности данных:
  - WrappedWarehousedFile;
  - WrappedWarehousedFileContents;
  - WrappedWarehousedFileMetadata;

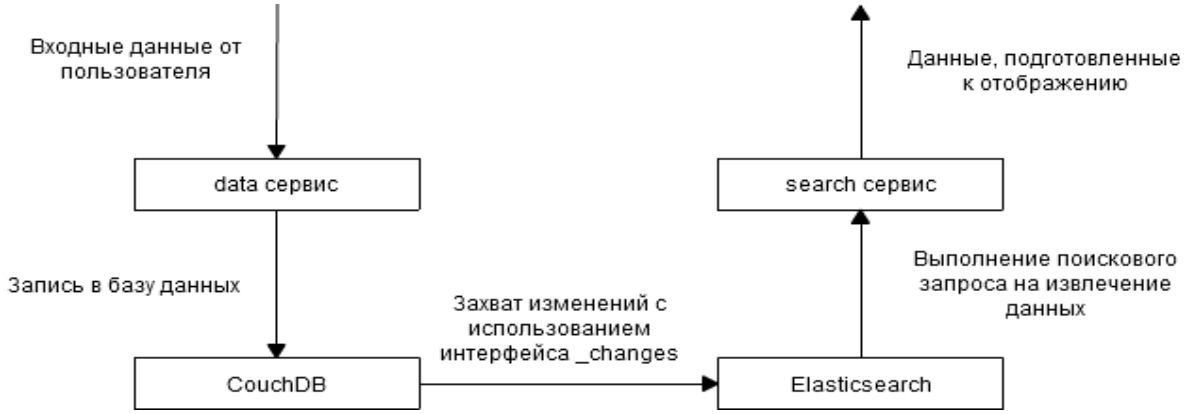
- Различные классы из search-model и search-elasticsearch, обрабатывающие поисковые запросы и строящие по ним результаты:
    - WarehousedFileDocumentRepository;
    - WarehousedFileContentsDocumentRepository;
    - WarehousedFileMetadataDocumentRepository;
    - BenchmarkedResultingPage;
    - HighlightedEntity;
    - Suggestions;
    - WarehousedFileSearchResults;
    - FindWarehousedFileByFullTextQueryRequest;
    - FindWarehousedFileByFullTextQueryResponse.

Результаты обработки запросов записываются в массив, что может занять некоторое количество времени. Для ускорения работы модуля можно поместить цикл, заполняющий архив данными, в отдельный поток. Это позволит начать строить страницу в процессе получения значений, что несомненно уменьшит время получения результатов поиска.

Аналогичным способом составляются классы, реализующие обработку других различных запросов, как поиск контактной информации, информации об аккаунтах и т.д.

3.3 Конфигурации

Сами данные попадают в поисковый сервер через интерфейс CoachDB “_changes” для получения внесенных изменений, в итоге сокращая количество передаваемых между компонентами сообщений.



Взаимодействие CouchDB и ElasticSearch

В результате, в базе данных содержатся эталонные записи, а в поисковой сервер поступают уже обработанные записи, пригодные для работы.

В ранних прототипах системы для индексации базы данных CouchDB использовался Elasticsearch плагин river-couchdb, однако в актуальной версии поискового сервера его убрали, и теперь необходимо использовать плагин Logstash.

По сути, он предоставляет схожий функционал, однако предыдущие плагины могли вызвать нестабильность в работе кластера, после чего было принято решение о замене river-couchdb на Logstash.

```
        input                                                    {
  couchdb_changes                                                {
   ignore_attachments                    =>                      true
   db                                    =>                      "wisefox"
   host                                  =>                      "localhost"
   port                                  =>                      5984
   filter                                =>                      null
   script                          =>                            "defineCouchDbDocumentType"
   script_type                           =>                      groovy
}
}
output                                                           {
elasticsearch                                                    {
   host                                  =>                      "localhost"
```

```
  index => "wisefox"
  idle_flush_time => 1
  protocol => "http"
  host => localhost
  port => 9200
  retry_max_interval => 10
 }
}
```

Конфигурационный файл Logstash

Важный вопрос стоит также и о подборе анализатора, подходящего под хранящиеся данные и поступающие запросы. В данном модуле используется следующий анализатор:

```
        {
  "settings" : {
    "analysis" : {
      "filter" : {
        "suggestion_shingle" : {
          "type":"shingle",
          "max_shingle_size" : 15,
          "min_shingle_size" : 2
        }
      },
      "analyzer" : {
        "suggestion_shingle" : {
          "filter" : [
            "lowercase",
            "suggestion_shingle"
          ],
          "tokenizer" : "standard"
        },
        "hierarchical_path" : {
          "tokenizer" : "path_hierarchy"
        }
```

```
        }
      }
    }
}
```

Настройки Elasticsearch

Рассмотрим его работу подробнее. В первую очередь задаётся фильтр “suggestion_shingle” типа “shingle”, что позволяет токенизатору использовать комбинацию токенов, как один токен, и устанавливается минимальный и максимальный размер этих токенов. Далее следуют параметры самого анализатора — в качестве фильтра используется ранее упомянутый “suggestion_shingle” и “lowercase”, который приводит полученный запрос к нижнему регистру. Токенайзер “path_hoerarchy” используется для разбиения адреса директории на токены.

Процесс индексации также является важной частью поискового модуля. Данные проходят предварительную обработку на поисковом сервере для соответствия с пользовательскими запросами. Для гибкой настройки индексации типовых сущностей используются JSON-файлы:

```
        {
  "WarehousedFile"                    :                    {
    "_parent"                         :                    {
      "type"                  :              "WarehouseDirectory"
    },
    "_routing"                        :                    {
      "required"                      :                    true
    }
  }
}
```

Пример схемы данных на поисковом сервере

Аналогично строятся остальные JSON-файлы, описывающие схемы данных, поступающих на поисковой сервер:

- Account — данные о аккаунте;
- AssignedRoles — идентификатор роли в системе;
- ContactInformation — контактная информация пользователя;
- WarehousedFile — данные о местоположении файла;
- WarehousedFileContents — содержимое поступающих файлов;

3.4 Тестирование

Тестируемые функции: выполнение полнотекстовых поисковых запросов для выборки документов.

Тестируемый элемент: выполнение полнотекстового поискового запроса с использованием инструментов поиска (формы поиска) на главной панели приложения.

Ввод: выбрать из списка атрибутов “Искать” элемент “везде”. В поле поиска ввести запрос, далее нажать кнопку поиска.

Вывод: Будут отображены результаты поиска, а именно документы, имеющие совпадения с запросом

Выводы

В ходе проделанной работы был реализован поисковой модуль, обеспечивающий поиск в распределенной системе хранения файлов. Предоставленный функционал можно считать необходимым для работы системы поиска в комплексе хранилища документов. Данная система проводит индексацию загруженных документов, производит полнотекстовый поиск по содержимому и поиск документов по необходимым атрибутам, а также ассинхронно выгружает полученные результаты. Что в совокупности делает систему действительно мощной и быстрой.

Отдельно стоит заметить, что полученная система отличается особой гибкостью в отношении настроек параметров поиска и индексации, путем корректировки конфигурационных файлов. В будущем дополнительный функционал по работе с запросами пользователей легко может быть добавлен, путем описания необходимых сущностей и последующей работе с ними по мере необходимости.

Заключение

Основная задача выпускной квалификационной работы — модернизация существующего прототипа системы хранения хранения текстовых файлов.

Во время работы с данной системой был изучен стек используемых технологий. Был проведен анализ различных готовых решений, приспособленных для работы с данным.

Ввиду высокой популярности данных технологий, можно утверждать, что выбранные решения являются долговечными, так как они имеют понятную и доступную документацию, качественную поддержку и большое сообщество.

При дальнейшей работе, функционал готового продукта можно расширять. Помимо работы над остальными модулями системы, может быть реализован такой полезный функционал, как синхронизация с мобильными устройствами.

По достижении финальной стадии разработки и готовности всех модулей системы, выпущенный продукт может использоваться на кафедре компьютерного моделирования и многопроцессорных систем для обеспечения удобного доступа к учебным материалам.

Список использованной литературы

1. Кападиа Амар, Варма Средхар, Раджана Крис Реализация облачного хранилища с Openstack Swift. Packt Publishing, 2014. 140 с.
2. The Apache Software Foundation. Maven – Introduction to the Build Lifecycle – https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
3. Broder, A. (2002). A taxonomy of Web search. SIGIR Forum, 36(2), 3–10.
4. Moore, Ross. "Connectivity servers". Cambridge University Press. http://nlp.stanford.edu/IR-book/html/htmledition/connectivity-servers-1.html
5. Official Website http://www.elasticsearch.org/
6. "DB-Engines Ranking - popularity ranking of search engines" htttp://db-engines.com/en/ranking/search+engine
7. elasticsearch Guide: Gateway http://www.elasticsearch.org/guide/reference/modules/gateway/
8. Elasticsearch as database http://karussell.wordpress.com/2011/07/13/jetslide-uses-elasticsearch-as-database/
9. Apache Solr vs Elasticsearch The Feature Smackdown http://http://solr-vs-elasticsearch.com/

Приложение

Приложение 1. Листинг WarehousedFileSearcherService

```
1. package ru.spbu.cc.wisefox.search.elasticsearch.service.implementation;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.stereotype.Service;
8.
9. import ru.spbu.cc.wisefox.domain.model.entity.WrappedWarehousedFile;
10. import ru.spbu.cc.wisefox.domain.model.entity.WrappedWarehousedFileContents;
11. import ru.spbu.cc.wisefox.domain.model.entity.WrappedWarehousedFileMetadata;
12.
13. import
14. ru.spbu.cc.wisefox.search.elasticsearch.repository.behaviour.WarehousedFileContentsDoc
15. mentRepository;
16. import
17. ru.spbu.cc.wisefox.search.elasticsearch.repository.behaviour.WarehousedFileDocumentRe
18. ository;
19. import
20. ru.spbu.cc.wisefox.search.elasticsearch.repository.behaviour.WarehousedFileMetadataDoc
21. mentRepository;
22.
23. import ru.spbu.cc.wisefox.search.model.behaviour.RemoteWarehousedFileSearcher;
24.
25. import ru.spbu.cc.wisefox.search.model.entity.BenchmarkedResultingPage;
26. import ru.spbu.cc.wisefox.search.model.entity.HighlightedEntity;
27. import ru.spbu.cc.wisefox.search.model.entity.Suggestions;
28. import ru.spbu.cc.wisefox.search.model.entity.WarehousedFileSearchResults;
29. import
30. ru.spbu.cc.wisefox.search.model.entity.request.FindWarehousedFileByFullTextQueryRequ
31. st;
```

```
32. import
33. ru.spbu.cc.wisefox.search.model.entity.response.FindWarehousedFileByFullTextQueryRes
34. onse;
35.
36. @Service
37. public class WarehousedFileSearcherService implements
38.         RemoteWarehousedFileSearcher {
39.
40. @Autowired
41. private WarehousedFileContentsDocumentRepository
42. warehousedFileContentsDocumentRepository;
43.
44. @Autowired
45. private WarehousedFileDocumentRepository warehousedFileDocumentRepository;
46.
47. @Autowired
48. private WarehousedFileMetadataDocumentRepository
49. warehousedFileMetadataDocumentRepository;
50.
51. public FindWarehousedFileByFullTextQueryResponse getByFullTextRequest(
52.         FindWarehousedFileByFullTextQueryRequest request) {
53.         BenchmarkedResultingPage< HighlightedEntity<WrappedWarehousedFileContents>
54. > page =
55.         this.warehousedFileContentsDocumentRepository
56.                 .findHighLightsByFullTextQuery(
57.                         request.getQuery(),
58.                         request.getPagination());
59.
60.         FindWarehousedFileByFullTextQueryResponse response =
61.         new FindWarehousedFileByFullTextQueryResponse();
62.
63.         List<String> corrections =
64.         this.warehousedFileContentsDocumentRepository
65.                 .getQueryCorrectionVariants(
66.                         request.getQuery());
67.
68.         List<WarehousedFileSearchResults> searchResults =
```

```
69.             new ArrayList<WarehousedFileSearchResults>();
70.
71.             Thread searcherThread = new Thread (new Runnable() {
72.
73.             public void run() {
74.
75.             for (HighlightedEntity<WrappedWarehousedFileContents> fileContents :
76. page) {
77.                     WarehousedFileSearchResults result =
78.                             new WarehousedFileSearchResults();
79.
80.                     WrappedWarehousedFileContents contents = fileContents.getEntity();
81.
82.                     WrappedWarehousedFile file =
83. this.warehousedFileDocumentRepository.findOneByIdAndRouting(
84.                             contents.getParent(), contents.getRouting());
85.
86.                     WrappedWarehousedFileMetadata metadata =
87.                 this.warehousedFileMetadataDocumentRepository.findOneByParentIdAndRouting(
88.                             contents.getParent(), contents.getRouting());
89.
90.                     result.setHighlights(fileContents);
91.                     result.setFile(file);
92.                     result.setMetadata(metadata);
93.                     searchResults.add(result);
94.                     }
95.             }
96.             });
97.
98.             searcherThread.start();
99.
100.                    response.setSuggestions(new Suggestions(corrections));
101.                    response.setPage(new BenchmarkedResultingPage<WarehousedFileSearchResults>(
102.                    page.getElementsTotal(),
103.                    page.getPagesTotal(),
104.                    page.getCurrentPageNumber(),
105.                    page.getPageSize(),
```

```
106.                page.getTimeTook(),
107.                searchResults));
108.                return response;
    a.  }
109.
110.        }
```